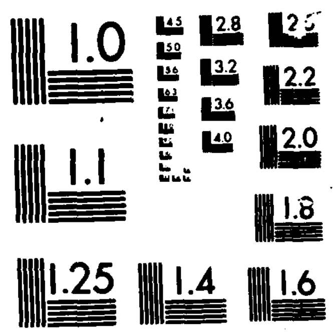
AD-A166 083 DISTRIBUTED DATA STRUCTURES: A CASE STUDY(U) ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE C S ELLIS AUG 85 TR-150 N00014-82-K-0193 1/1

UNCLASSIFIED F/G 9/2 NL



MICROCOP' CHART

AD-A166 083

(12)

Distributed Data Structures: A Case Study

Carla Schlatter Ellis
Computer Science Department
University of Rochester
Rochester, NY 14627

TR150
August, 1985

Department of Computer Science
University of Rochester
Rochester, New York 14627

(A-

Distributed Data Structures: A Case Study

Carla Schlatter Ellis
Computer Science Department
University of Rochester
Rochester, NY 14627

TR150
August, 1985

Abstract

In spite of the amount of work recently devoted to distributed systems, distributed applications are relatively rare. One hypothesis to explain this scarcity of different examples is a lack of experience with algorithm design techniques tailored to an environment in which out-of-date and incomplete information is the rule. Since the design of data structures is an important aspect of traditional algorithm design, ~~we feel that~~ it is important to consider the problem of distributing data structures. In this paper, we investigate these issues by developing a distributed version of an extendible hash file which is a dynamic indexing structure that could be useful in a distributed database.

The preparation of this paper was supported in part by the National Science Foundation, under Coordinated Experimental Research grant No. DCR-8320136 and under grant No. MCS-8203306, and in part by the Defense Advanced Research Projects Agency, monitored by the ONR, under grant No. N00014-82-K-0193.

1. Introduction

There is currently a significant amount of work being done in the area of distributed systems. Among the motivations usually cited for the use of a distributed system are ease of expansion, increased reliability, actual geographic distribution, the ability to incorporate heterogeneous resources, and resource sharing among autonomous sites. In spite of this, distributed *applications* are relatively rare. By this we mean problems that actually *exploit* some aspect of distribution and have been solved by user-level distributed programs. As an example, one can easily imagine problems requiring the computational power of a supercomputer along with an attractive user interface using the window package of a personal workstation that would benefit from the ability to incorporate both kinds of machines into the solution. There are a number of hypotheses to explain the scarcity of examples including inadequate performance in networks and lack of programming language support. A more important problem may be a lack of experience with algorithm designs that tolerate inaccurate and inconsistent data. It appears to be a fundamental characteristic of distributed computations that no one component can easily gather knowledge of the true instantaneous global state of the system. Thus, out-of-date and incomplete information is inevitable. The purpose of our research has been to investigate distributed programming techniques that acknowledge this principle.

Since the design of the data structures is an important aspect of traditional algorithm design, we feel that it is valuable to consider the problem of distributing data structures. For our purposes, a distributed system is modeled as a number of logical processors communicating solely through port-based asynchronous message-passing in the style of [Rashid 80]. There is no memory shared among these logical processors. A logical processor may encompass multiple processes that execute on the same physical processor and may share data among themselves. The phrase "distributing a data structure" means that there are a number of logical processors each encapsulating some portion of a single coherent data structure and acting as a manager for that piece. The data structure may either be divided into disjoint portions or some parts may be replicated in several managers. Replication may serve to increase availability of the data structure when processors can fail or to improve performance by allowing more concurrency through a bottleneck of the structure or by placing copies of heavily used information at user's sites. Such replication raises the issue of maintaining consistency to an appropriate degree. Although a number of general purpose mutual consistency algorithms are available [Gifford 79, Stonebraker 79, Thomas 79], often it should be possible to exploit certain properties of the specific problem at hand to arrive at a less synchronized method. In this paper, we investigate these issues by developing a distributed version of a particular indexing structure.

2. A Distributed Version of Extendible Hashing

Hashing has long been recognized as a fast method for accessing records by key in large relatively static databases. However, when the amount of data is likely to vary significantly, traditional hashing can suffer from performance degradation and may eventually require rehashing all the records into a larger space. Extendible hashing [Fagin 79] is one of a number of recently developed hashing schemes [Larson 78, Litwin 80, Lomet 83, Litwin 78] that can grow and shrink in response to insertion and deletion operations. A distributed system can provide the growth in resources to accommodate such growth in the data structure. Thus it makes sense to investigate how to partition an extendible hash file among the sites in a distributed environment. In addition, availability considerations demand that any data structure used as an index for a distributed database be itself distributed and possibly replicated. Finally, it appears to be relatively easy to distribute components of an extendible hash file in such a way that operations involve as few sites as possible.

The sequential algorithms for extendible hashing are described in [Fagin 79]. The basic ideas and terminology are summarized below. The data structure consists of two parts: a set of *buckets* and the *directory*. The buckets reside on secondary storage and contain keys and associated information. The order of the data within buckets is not important for this discussion. The directory is an array of pointers to buckets. A hash function is used that generates a very long *pseudokey* when applied to a key. The number of bits of the pseudokey actually used to index into the directory is called the *depth* of the directory and changes as the file grows or shrinks. In our work, the least significant bits are used in order to simplify manipulations of the directory. Suppose that the directory's depth is currently three. This means that at the moment, there are eight valid directory entries. The i^{th} entry, $0 \leq i \leq 7$, points to the bucket that holds all the records whose pseudokeys end in the three bit binary representation of i. Each bucket includes a *localdepth* ($\leq$ depth) indicating that the pseudokeys of the records it contains agree in only that number of bits. Thus multiple directory entries will point to the same bucket if its localdepth is less than the directory's depth. Figure 1 gives an example of an extendible hash file for sequential access. To perform a find operation for a key, k, one would apply the hash function to k to obtain the pseudokey (imagine it is '...101'), determine the current depth of the directory (2 in this example), and use the appropriate bits ('01'), as an index. Following the pointer in the directory entry, one would search the third bucket for k. As insertions occur, a bucket may become full (indicated by the *count* field) and split into two buckets. If the old localdepth equals depth, the directory doubles in size and depth increases by one. Similarly, deletions may result in two buckets merging and possibly reducing the depth of the directory. One way of detecting the condition that allows halving the size of the directory is to keep a count (named *depthcount*) of the number of buckets whose localdepth equals depth. Figure 2 shows how a sequence of updating operations would affect the structure given in Figure 1 where $x < y = z =$ maximum number of keys allowed in a bucket. This data structure is our point of departure for developing a distributed solution. The obvious partitioning calls for two types of logical processors, namely directory managers that are responsible for replicas of the directory component and bucket managers. Each bucket manager is responsible for a disjoint subset of the buckets.

The distributed solution is derived from a solution allowing concurrent access to a shared centralized extendible hash file [Ellis 83]. That solution is based on locking protocols and modifications in the data structure to allow for concurrency. Additional modifications are introduced here to improve locality and allow replication of the directory component. The fundamental change from the sequential version is that the buckets are linked through a *next* field to allow recovery from concurrent restructuring operations. This provides an alternate path to the desired data that can

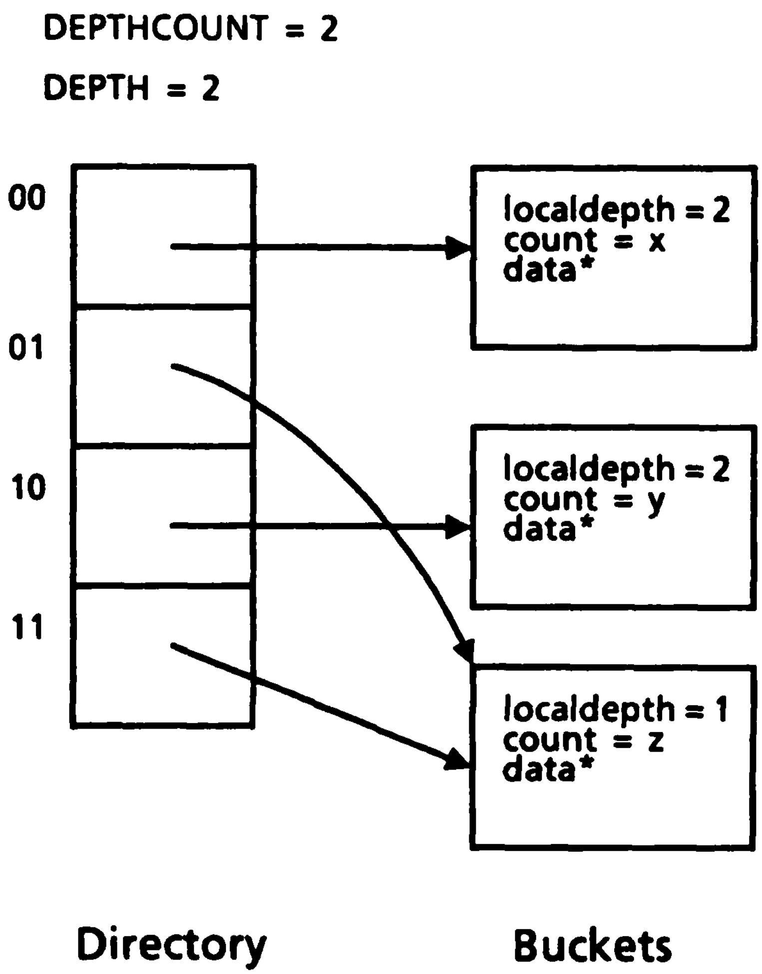
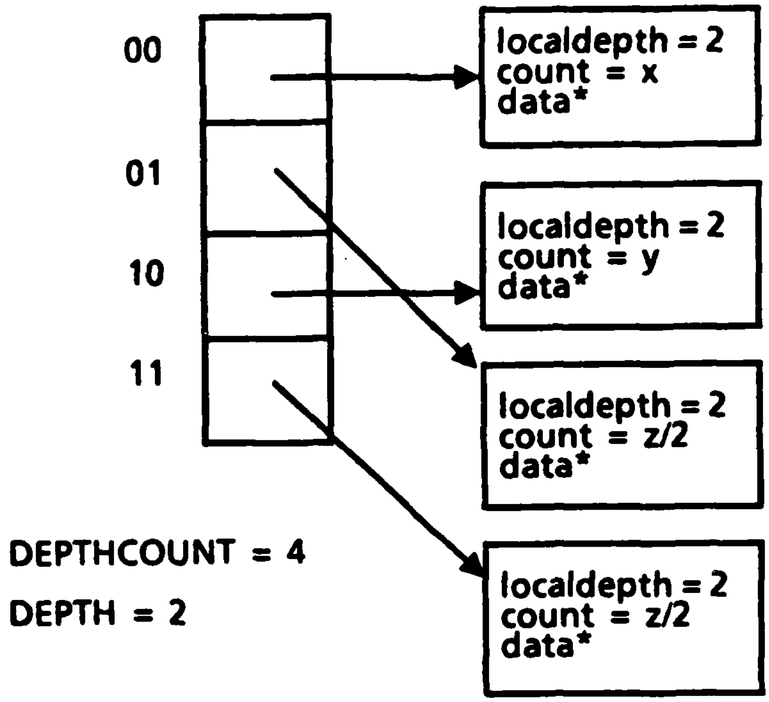
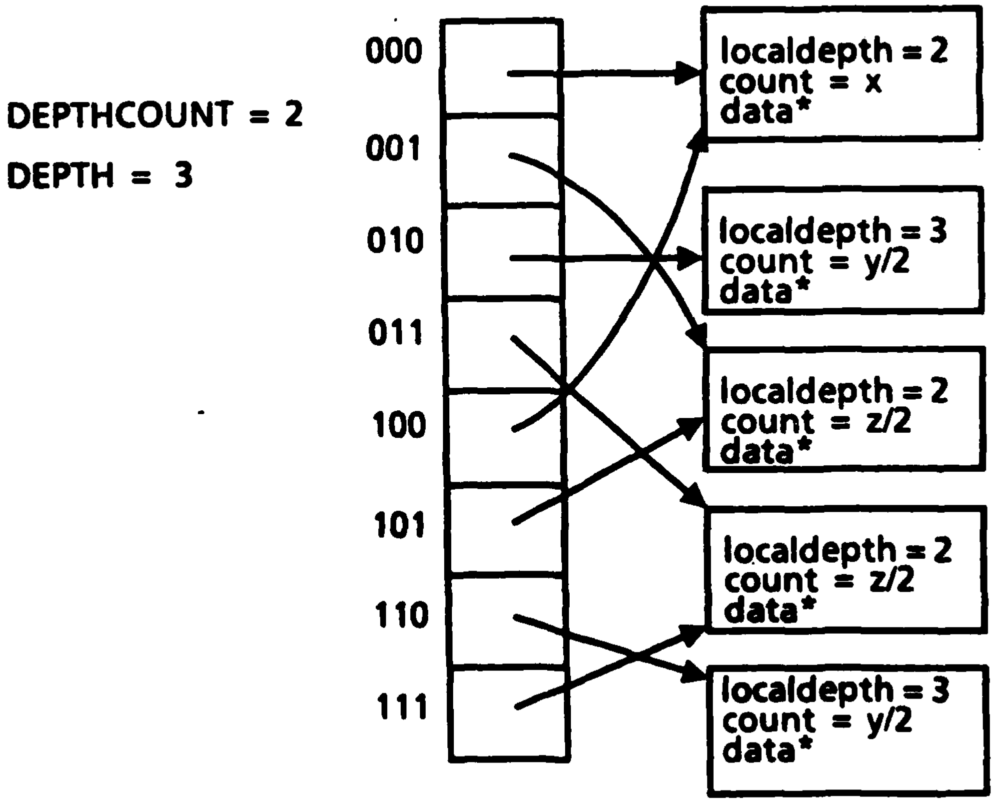


Figure 1 Sequential Access Extendible Hash File



a) After inserting record with pseudokey ...11 into Fig. 1, causing split.



b) After inserting record with pseudokey ...10, causing split and doubling of directory.

Figure 2 Updates

be used by a searching operation when the information is being moved in a split or merge operation. The approach is similar to the use of link pointers in Lehman and Yao's B^{link}-tree solution [Lehman 81]. When a bucket splits, the next link of the original bucket is reassigned to point to the newly created bucket. The new bucket gets the original bucket's old next pointer. Merging does the reverse. The next pointer is also used for recovery through deleted, but not yet deallocated, buckets. Deleted buckets and discarded halves of the directory are actually deallocated only after ensuring that they are no longer needed. In addition, there must be a way for a bucket manager performing the search phase of a transaction to tell if it has read the wrong bucket. We chose to include a field (*commonbits*) containing the common bit pattern that characterizes the pseudokeys that belong in the bucket. Alternatively, one could reapply the hash function to any key stored in the bucket and use this for comparison with the target pseudokey as long as the possibility of an empty bucket is taken care of. "Wrong bucket" includes the case where the bucket has been merged into a preceding bucket. That bucket is marked as "deleted" (using commonbits field). A *prev* link has been added to each bucket that leads to the bucket from which this bucket originally split off. This information which is local to the bucket manager is used to simplify finding the partner bucket for a possible merge. Each link now represents a pair consisting of a long-lived identifier for a manager port and a bucket address that is meaningful to that manager. A *version* field introduced into each bucket and each directory entry is used in updating directory copies asynchronously. The resulting data structure appears in Figure 3. Two copies of the directory are shown in that figure. Note that this example represents a consistent state with no update operations in progress.

The main purpose behind the modifications is to make it possible to tolerate inconsistencies and inaccuracies in the directory data. In order to gain some intuition for these structural changes, consider the configuration shown in Figure 4. There are two active update operations: an insertion of a record with pseudokey '.... 00' that has just caused a split and the deletion of the only record left with pseudokey of the form '.... 11' causing a merge. The top copy of the directory has not yet recorded the effect of the split and the bottom copy does not yet reflect the merge. Suppose there is a find operation for pseudokey '.... 10' directed at the topmost directory. The first bucket retrieved is the wrongbucket as indicated by the comparison of the pseudokey and commonbits and the search continues with the next bucket which turns out to be the desired one. Similarly, consider a search for pseudokey '.... 11' directed at the bottom copy of the directory. The first bucket read is marked as deleted and the next link leads to the appropriate bucket.

The actions taken by the managers in response to messages received are discussed below. Figure 5 shows the message types that flow between the various managers. The information contained in these messages is outlined in Figure 6. A condensed version of the procedure for the directory manager, written in a C-like syntax [Kernigan 78], is given in Figure 7. The directory manager is presented here as a server capable of handling multiple user requests. The bucket manager is written as a front end process that serves as the initial contact for its set of buckets and a set of associated processes that reside at the same site and share secondary memory. The pseudo-code for these processes is given in Figure 8.

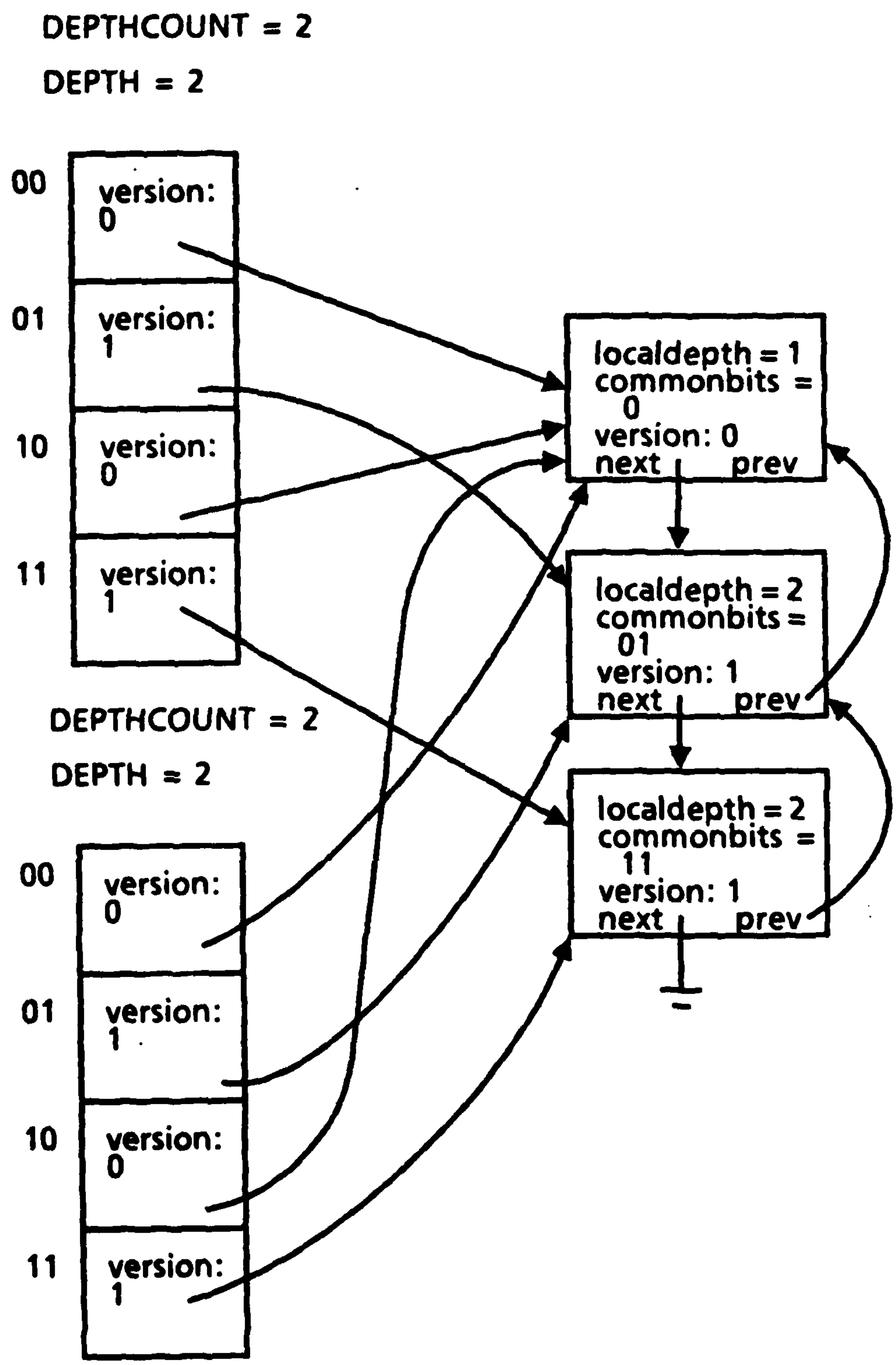


Figure 3 Distributed Extendible Hash File

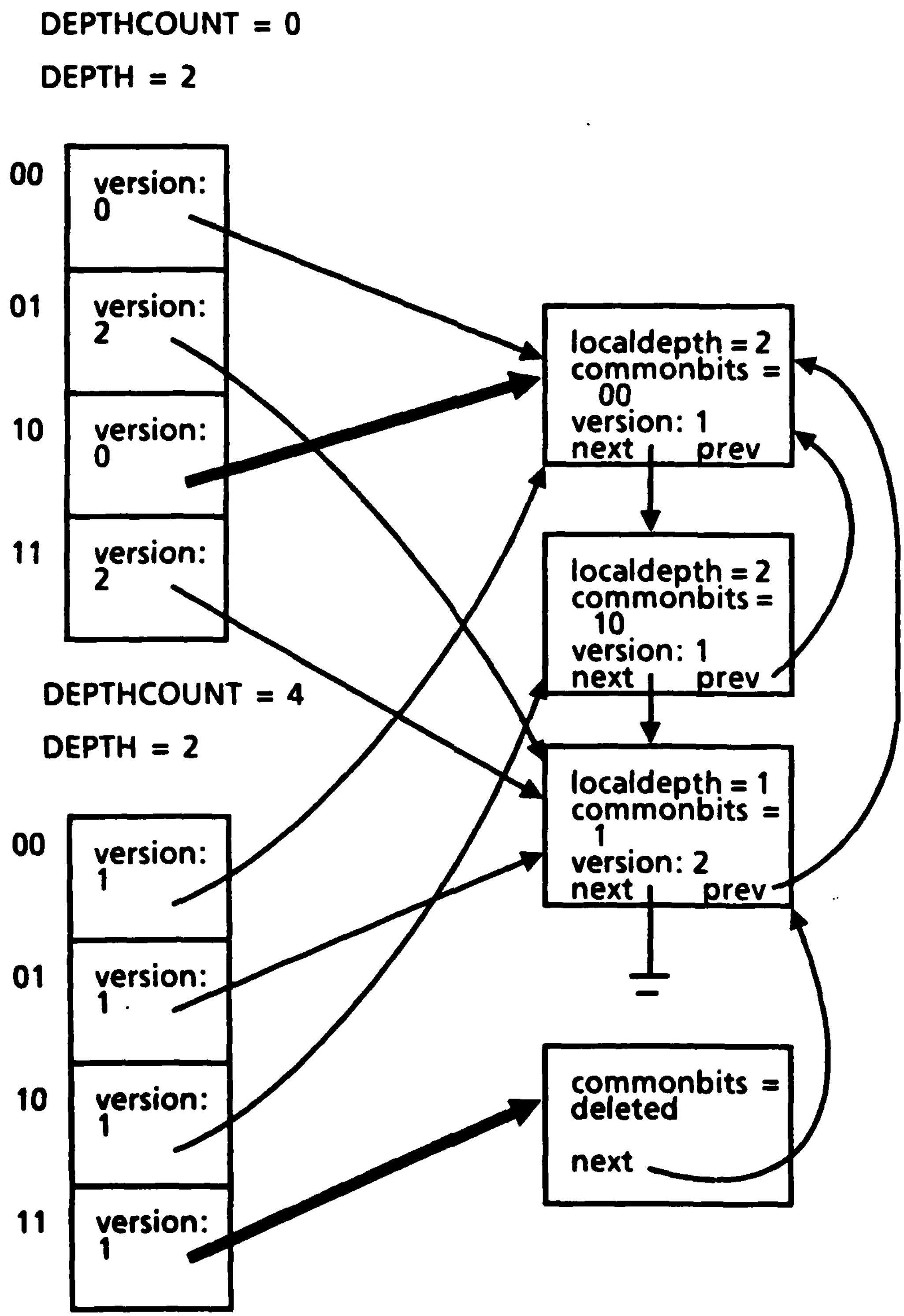


Figure 4 Out of Date Distributed Hash File

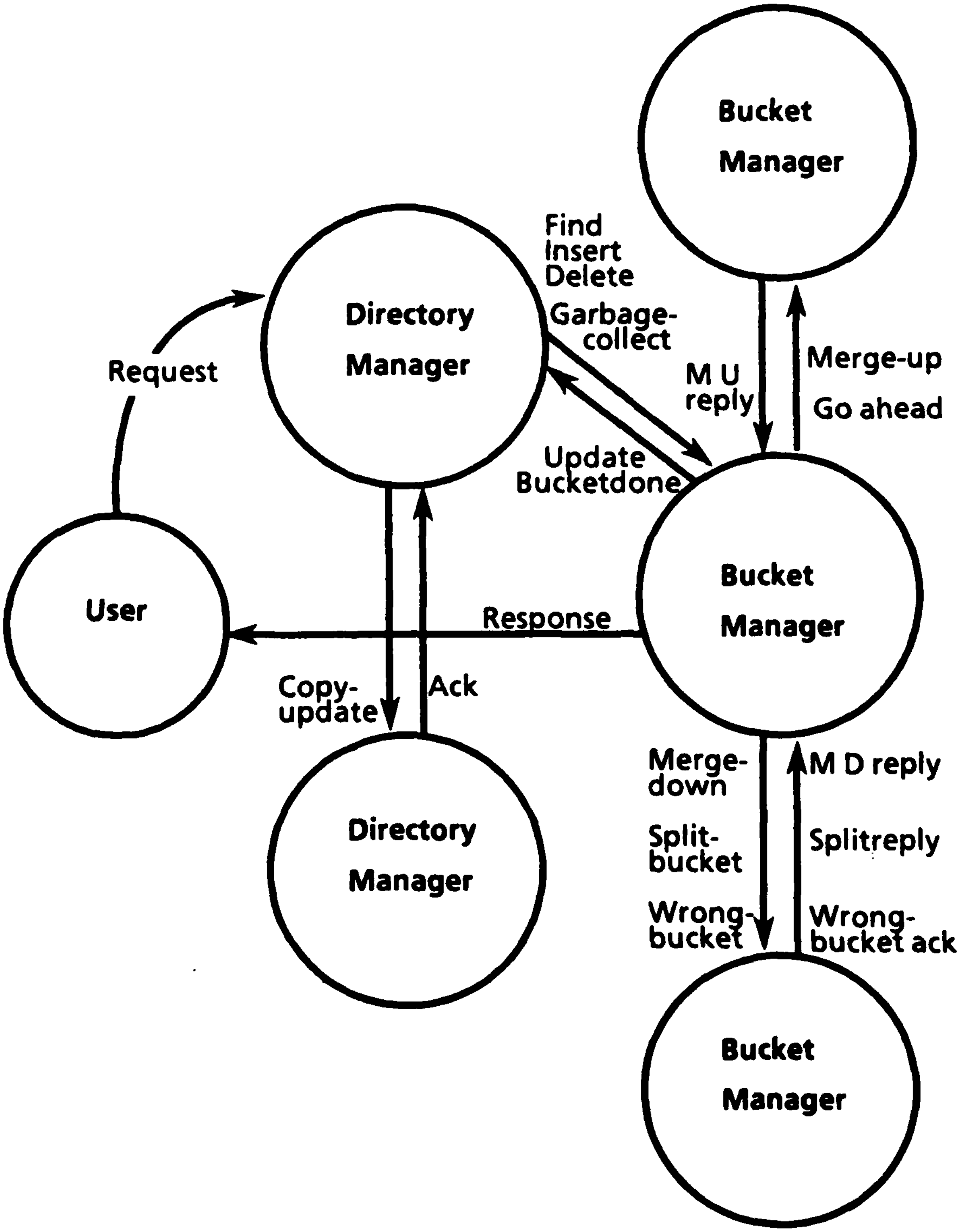


Figure 5 Managers and Message Flow

message id	data in message
Request	desired key op: (find | insert | delete) user's port
User Response	success: (true | false)
Find, Insert, Delete	desired key transaction # page address user's port directory manager's reply port pseudokey user's reply port
Bucketdone	transaction # success: (true | false)
Update	transaction # old localdepth version # of "0" partner version # of "1" partner new page address id of bucket manager success: (true | false)
Copy update	op: (insert | delete) pseudokey old localdepth version # of "0" partner version # of "1" partner new page address id of bucket manager acknowledgement port
Ack for Copy update	
Garbage Collect	list of page addresses

message id	data in message
Wrongbucket	op: (find | insert | delete) desired key transaction # page address user's port directory manager's reply port pseudokey bucket manager's reply port
Ack for Wrongbucket	
Splitbucket	manager's reply port buffer contents of new half
Splitreply	new page address id of bucket manager
Mergedown	partner's address localdepth bucket manager's reply port
M.D. Reply	buffer contents success: (true | false)
Mergeup	partner's address bucket manager's reply port target bucket's address bucket manager's id
M.U. Reply	localdepth version # bucket manager's reply port success: (true | false)
Go ahead	next link next bucket manager id version # success: (true | false)

Figure 6 Messages

Figure 7 Pseudocode for Directory Managers

Notation

C-like statements;
English-like pseudocode statements;
/*comments*/

```
while ( true ) {
    messageid = GetMessage (&msg);
    /* Either receives a message or takes a message off
    the list of delayed but now ready directory updates. */
    switch (messageid) {
        case request: /* from user */
            readcount = readcount + 1;
            /*number of transactions in progress*/
            Calculate pseudokey and locate current
            incarnation of the bucket manager responsible
            for desired bucket;
            Generate transaction # and save state related
            to this request;
            Construct and send a "find", "insert" or
            "delete" message;
            break;

        case bucketdone: /* from bucket manager -
        no directory update needed */
            RestoreState (msg.transaction #);
            /* Recall context for this request */
            if (!msg.success && operation = = delete) {
                Try again: locate bucket manager again
                and reissue "find", "insert" or "delete"
                message;
            }
            else {
                readcount = readcount - 1;
                CleanState (msg.transaction #);
                /* forget about this request */
            }
            break;

        case update: /* directory update at directory manager
        that initially handled request */
            RestoreState (msg.transaction #);
            Send a copyupdate message to all other directory
            managers and increment copycount for each
            outstanding directory update
            if (VersionsDoNotMatch(msg))
            /* compares version numbers in message with
            version numbers in corresponding directory entries */
                DelayUpdate(msg);
            else {
                if ( operation = = insert) {
                    Apply appropriate updates
                    to local copy of directory;
                    if (!msg.success) {
                        Try again;
                    }
                    else {
                        readcount = readcount - 1;
                        CleanState (msg.transaction #);
                    }
                }
                else { /* op = delete */
                    Record location of deleted bucket for the
                    eventual garbage collection phase;
                    Apply local directory updates;
                }
                ReleaseSaved();
                /*If finishing this directory update enables previously
                delayed ones, make them accessible to GetMessage */
            }
            break;

        case copyupdate: /* from other directory managers */
            if (VersionsDoNotMatch(msg))
                DelayUpdate(msg);
            else {
                if (msg.op = = insert) {
                    Apply local directory updates;
                    SendAck(msg.ackport); /* respond to
                    directory manager who initiated this update */
                }
                else { /* op = delete */
                    Apply local directory updates;
                    RememberAck(msg.ackport);
                    /* save up acks until the equivalent of
                    exclusive-locking occurs */
                }
                ReleaseSaved();
            }
            break;

        case ack : copycount = copycount - 1;
            break;
    }
    if (!readcount) SendRememberedAcks();
    /*send acks saved by deletion copyupdates */
    if (!readcount && !copycount) GarbageCollect();
    /*get rid of buckets deleted through this directory manager */
}
```

Figure 8 Pseudocode for Bucket Managers

Bucket Manager Front End Process:

```
while (true) {
    messageid = receivemessage (&msg);
    if (messageid = = splitbucket) { /* from another bucket
    manager with no available space */
        Allocate available page;
        putbucket (newpage, msg.half2);
        Send "SplitReply" message containing link to new bucket;
    }
    else {
        Create a bucket slave process and forward msg to it;
    }
}
```

Bucket Slave Process:

```
messageid = receivemessage (&msg);
if (messageid = = wrongbucket) sw = msg.op;
else sw = messageid;
switch (sw) {

    case find: oldpage = msg.page;
        ReadLock (oldpage);
        if (messageid = = wrongbucket)
            Send "Ack" to bucket manager holding
            previous bucket; /*allows it to unlock */
        else Send successful "Bucketdone" message;
          /*tells directory manager that no update is needed */
        getbucket (oldpage, current);
        onmachine = true;
        /*Follow next links until current is the right bucket:*/
        while (current is wrong bucket && onmachine) {
            newpage = current -> next;
            machine = current -> nextmgr;
            if (machine != me) { /* next bucket is remote */
                Send "Wrongbucket" message to next bucket
                manager;
                onmachine = false;
            }
            else { /* next bucket is local */
                ReadLock (newpage);
                getbucket (newpage, current);
                UnReadLock (oldpage);
                oldpage = newpage;
            }
        }
        if (onmachine) {
            if (search (current, msg.key))/* is key there? */
                found (msg.key);
            else
                notfound (msg.key);
        }
        else receivemessage (&msg); /* Wrongbucket reply */
        UnReadLock (oldpage);
        break;

    case insert: onmachine = true;
        oldpage = msg.page;
        SelectiveLock (oldpage);
        if (messageid = = wrongbucket)
            Send "Ack" to previous bucket manager;
        getbucket (oldpage, current);
        Follow next links until current is the right bucket
        (as in find case except use Selective locks instead of
        Read locks);
        if (!onmachine) {
            receivemessage (&msg); /* Wrongbucket reply */
            UnSelectiveLock (oldpage);
        }
        else {
            if (search (current, msg.key)) {/*is key already there?*/
                success = true;
                Send "Bucketdone" message;
                UnSelectiveLock (oldpage);
            }
            else if (current -> count != numentries) {
                /* current bucket not full */
                success = true;
                Send "Bucketdone" message;
                add (current, msg.key);
                /*inserts key into current buffer */
                putbucket (oldpage, current);
                UnSelectiveLock (oldpage);
            }
            else {/*current is full - directory will be affected */
                success = split (current, half1, half2, msg.key);
                /*distributes the contents of the current bucket into
                2 buffers pointed to by half1 and half2;
                if room available, inserts key into appropriate half
                and returns true; otherwise returns false */
                if (AvailablePages()) {
                    newpage = allocbucket ();
                    machine = myid;
                    putbucket (newpage, half2);
                }
                else {/* no available pages locally */
                    Send "Splitbucket" message
                    containing contents of new bucket
                    to a manager with space;
                    receivemessage (&msg); /*split bucket reply*/
                    machine = msg.bucketmgr;
                    newpage = msg.page;
                }
                half1 -> next = newpage;
                half1 -> nextmgr = machine;
                putbucket (oldpage, half1);
                UnSelectiveLock (oldpage);
                Send "Update" message to originating
                directory manager telling it to update directory;
            }
        }
        break;
```

Figure 8 (continued)


```
case delete:
    Find the right bucket as in the beginning of insert
     except place Exclusive locks;
    if (!onmachine) {
        receivemessage (&msg); /* Wrongbucket ack */
        UnExclusiveLock (oldpage);
    }
    else {
        if (current bucket will not be left "too empty"
          as a result of deleting msg.key) {
            Send successful "Bucketdone" message;
            if (remove (msg.key, current))
                putbucket (oldpage, current);
            UnExclusiveLock (oldpage);
        }
        else { /*Merging partner buckets is called for*/
            if (msg.key is in first bucket of the pair) {
                newpage = current -> next;
                machine = current -> nextmgr;
                if (machine = me) {
                  Merge on site;
                }
                else {/* partner is remote */
                    Send "Mergedown" message to
                    partner's bucket manager;
                    receivemessage (&msg);
                    /* Mergedown Reply expected */
                    if (msg.success) {/*OK to merge
                    (i.e. localdepths match);
                    contents of partner in msg */
                        Construct merged bucket
                        in current buffer;
                        putbucket (oldpage, current);
                        Send successful "Update" message;
                    }
                    else {/* simply remove record */
                        Send successful
                        "Bucketdone" message;
                        if (remove (z, current))
                            putbucket (oldpage, current);
                    }
                    UnExclusiveLock (oldpage);
                }
            }
            else { /* msg.key in second of pair */
                newpage = current -> prev;
                machine = current -> prevmgr;
                UnExclusiveLock (oldpage);
                if (machine = = me) {
                  Merge on site;
                }
                else {/* partner is remote */
                    Send "Mergeup" message to
                    partner's bucket manager;
                    receivemessage (&msg);
                    /* MergeUp Reply expected */
                    if (!msg.success) {
                    /* not mergable - simply remove record */
                        Send successful
                        "Bucketdone" message;
                        if (remove (z, current))
                            putbucket (oldpage, current);
                    }
                    else {/* apparently mergable from
                    partner's point of view- check more locally */
                        ExclusiveLock (oldpage);
                        getbucket (oldpage, current);
                        if (key to be deleted no longer
                        belongs in current bucket) {
                            UnExclusiveLock (oldpage);
                            Send "Goahead" message to partner
                            with success field set to false;
                            /*cancels merge */
                            Send "Bucketdone" message
                            with success = false;
                            /*tell directory manager to retry */
                        }
                        else if (current->localdepth
                          does not match localdepth in msg ||
                          current no longer "too empty") {
                            Send successful "Bucketdone"
                            message;
                            if (remove(z, current))
                                putbucket (oldpage, current);
                            UnExclusiveLock (oldpage);
                            Send "Goahead" message
                            with success = false;
                            /*cancel merge */
                        }
                        else {
                            Send successful "Goahead"
                            message to partner;
                            /*tell partner's manager to merge */
                            current -> next = current -> prev;
                            current -> nextmgr =
                                current -> prevmgr;
                            current -> commonbits = deleted;
                            putbucket (oldpage, current);
                            Send successful "Update"
                            message;

    } } } } } } }
    break;

case mergedown: newpage = msg.partner;
    ExclusiveLock (newpage);
    getbucket (newpage, brother);
    success = brother -> localdepth = = msg.localdepth;
    Send "MergeDown Reply" to partner;
    if (success) {
        brother -> commonbits = deleted;
        brother -> next = brother -> prev;
        brother -> nextmgr = brother -> prevmgr;
        putbucket (newpage, brother);
    }
    UnExclusiveLock (newpage);
    break;

case mergeup: newpage = msg.partner;
    ExclusiveLock (newpage);
    getbucket (newpage, brother);
    success = (brother -> next = = msg.target) &&
    (brother -> nextmgr = = msg.managerid);
    Send "MergeUp Reply";
    if (success) {
        receivemessage (&msg); /* "GoAhead" expected */
        if (msg.success) { /* merge */
            Construct merged bucket in brother;
            putbucket ( newpage, brother);
        }
    }
    UnExclusiveLock (newpage);
    break;

case garbagecollect:
    for each page in msg.list {
        ExclusiveLock (page);
        deallocate (page);
        UnExclusiveLock (page);
    }
    break;
}
```

In the centralized solution, the directory component was locked during the search for the target bucket to prevent interference between searches and deletions. A deleting process placed an incompatible lock. If the deleter did not exclude the reader and was in the process of halving the directory, the reader might have attempted to access an invalid directory entry based on the old value of depth. A similar interference could occur between readers and deleters with regard to recently deallocated buckets. The locking of the directory in the centralized solution translates into the manager's explicit scheduling of requests for its attention in the distributed version.

A user wishing to perform an operation on the distributed hash file may contact any directory manager with a *request* message. Upon receiving the request, the manager saves some state about the desired operation, does the directory lookup, and forwards the request to the bucket manager indicated. After forwarding the request, the directory manager can service another message. While a request is outstanding, the manager delays deallocation of deleted components that the request may be depending upon.

The forwarded request is eventually received by the bucket manager front end. A new slave process is created for each request requiring service from the bucket manager (with the exception of an off-site split which is handled by the remote front-end). The slave processes associated with a bucket manager can manipulate the data in buckets belonging to this manager after locking the bucket and transferring the information into private buffers. The buckets are assumed to occupy physical pages on disk which are read and written as single operations. The locking protocol uses various types of locks placed on individual buckets. The compatibility of lock types is given by the following table.

Lock request	Existing lock		
	read-lock	selective-lock	exclusive-lock
read-lock	yes	yes	no
selective-lock	yes	no	no
exclusive-lock	no	no	no

If the request message calls for a find operation, a read-lock is placed on the target bucket. For an insert operation, the slave process places an selective-lock and for a delete, an exclusive-lock.

Upon reading the data, the process may discover that it has the wrong bucket. This means that a split or merge has occurred that was not yet reflected in the copy of the directory that was read. In other words, now the localdepth low order bits of the target pseudokey do not match the commonbits of this bucket. By following the next pointer, the right bucket will eventually be found. The next bucket is always locked prior to releasing the lock on the current bucket. This flow of locks, known as *lock-coupling*, prevents processes from leapfrogging each other. If the next bucket belongs to a different bucket manager, a *wrongbucket* message is sent and acknowledged before the lock is released. Once the right bucket is found, the desired operation is performed and finally a response sent to the directory manager that initially handled the request. Lock incompatibilities prevent interference among updates. An insert or delete operation may result in a splitting or merging of buckets. Off-site splitting may be necessary if there is a shortage of available buckets locally. Off-site merging occurs when the partner bucket belongs to a different manager. Protocols are available to handle these situations (*splitbucket*, *mergedown*, and *mergeup* messages and associated replies). If a merge operation appears to be appropriate, the partner bucket can be determined using local information (i.e. either *next* or *prev* links). In the centralized algorithms it was acceptable to locate a partner bucket using the directory. In the distributed case, this would have involved additional message traffic for a bucket manager to send an inquiry message to a directory manager and wait for a reply. In order to avoid deadlock, the partners for a merge must be locked according to the ordering imposed by *next* links. If it is necessary to lock the bucket pointed to by *prev*, the lock on the target bucket must first be released and a number of conditions must be checked after gaining the locks. This results in the differences between the *mergeup* and the *mergedown* protocols.

Two possible responses may come back to the directory manager from a bucket manager, either *bucketdone* or *update*. Bucketdone will generally signify that no directory modifications are needed and the directory manager may now forget about this request. An update message calls for scheduling an update on the local copy according to version number and notifying all other directory managers by broadcasting a *copyupdate* message. For each outstanding unacknowledged remote directory modification, a counter is incremented that serves to prevent garbage collection. A bucket may not be deallocated until all directories send an *acknowledge* message. Upon receiving a copyupdate message, a directory manager schedules the update on its local copy and when the changes have been applied (and in the case of delete operations, when no outstanding requests remain at this manager), acknowledgements are sent.

Because obsolete directory information is usable, the multiple copy update does not have to be strictly synchronized (in the sense of an atomic transaction). However, the ordering of different directory modifications due to operations on the *same bucket* should be the same across all copies and determined by the order in which the bucket operations are performed. Each split or merge changes the version numbers of the affected buckets. A split generates two buckets with version numbers one greater than that of the original bucket. A merge results in one bucket with a version number one larger than the maximum version of the two partners. The version number in each directory entry should match the version of the bucket it points to when the directory is competely up to date. Each directory manager applies the modifications indicated by an update or copyupdate message to its local copy when the version numbers of the affected directory entries match the version numbers in the message which reflect the versions of the buckets involved. This use of version numbers for scheduling updates enforces the desired ordering. The following example illustrates why this ordering approach is adopted. Suppose first a split operation is performed almost immediately followed by a merge involving those two buckets. Imagine a directory manager that hears about these updates in the opposite order and applies them. The directory update related to the merge would essentially have no effect since the split had not yet been processed. The subsequent update related to the split would result in directory entries leading to a deleted bucket. At this point the directory is usable since next links provide recovery. However, since it appears that both messages have been serviced, the deleted bucket could then be deallocated. This would leave that copy of the directory in a truly incorrect state from which recovery would be impossible.

Under the assumptions that processes do not fail, message buffering is sufficient to eliminate blocking on a send, and messages are reliably delivered, then this solution can be shown to be deadlock free and correct in the sense that requests are serializable in their externally observable behavior. Although extremely unlikely, the theoretical possibility of indefinite postponement does exist.

In discussing the correctness of this algorithm, we wish to separate the arguments concerning the replication of the directory from those about the basic protocols and processing that service the user's request. This allows us to view the replicated directories as a single global directory with certain desirable properties in later phases of the discussion. Intuitively, we need a statement to the effect that the information gathered from a directory access may not accurately reflect the current state of the hash file; but it is incorrect in such a way that next links provide adequate recovery. We now attempt to formalize this idea somewhat and then show that our multiple copy update strategy actually maintains this property. Throughout this presentation, the term *transaction* is used for the execution of a single find, insert, or delete operation as it moves thorugh various managers.

The version of the directory seen by a transaction can be expressed as one member of a set of schedules, S, that defines the state of the directory. For this to make sense, we must elaborate on the notion of a schedule. Consider the set, A, of all split, merge and remove (enabling garbage collection) actions resulting from update operations that have changed the bucket structure by the time of the directory access in question. For example, a delete request may require no directory modifications at all or it may generate a merge and subsequent remove that become members of A. There is a partial ordering imposed in these actions based on when bucket modifications are made. Specifically, if two operations affect the same bucket, then there is a relationship established between them. A schedule is a totally ordered subset of A that obeys the following constraint: The order of actions within the schedule must be consistent with the partial order. No individual schedule in the set S necessarily represents the timing of bucket modifications; but rather, it can be viewed as encoding a valid directory structure at some point during a possible execution sequence of the actions in A. An action is considered done and its effects incorporated into the directory when it appears in each schedule of S. All other actions are still in progress. In the case of a delete request that causes two buckets to be merged, the deleted bucket is not deallocated until the associated remove action is done so recovery through its next link is still possible. The point is that the appropriate next links are set up before the related split or merge action appears in any schedule of S and deleted buckets remain in place until all schedules include the relevant remove action. Consequently, any member of S represents usable information.

In the implementation of the replicated directories, each copy corresponds to one schedule in the set. The sequence of actions in a schedule indicate the order of directory updates applied to that copy. A split action signifies the local execution of the updatedirectory procedure and possibly doubledirectory. A merge action represents the execution of halvedirectory or updatedirectory. Remove denotes the equivalent of placing an exclusive-lock on the local copy (i.e. testing the readcount). Inclusion in the set A can be defined by the set of update messages that have been sent from bucket managers to directory managers. A sequence of these actions is an appropriate model for the state of a single copy since the corresponding code sections are performed serially by the manager. There are various ways of enforcing this requirement. In the multiplexed directory manager given, access to its copy of the directory by concurrent transactions is controlled by explicit scheduling. the receipt of a message establishes a context for the resulting processing and the directory structure is put into a consistent state before the context changes again. Either the required values are contained within the incoming message to initialize the context (e.g. copyupdate or request messages) or saved values that were previously tagged with a transaction number are restored when further steps must be taken on behalf of the transaction (e.g. due to arrival of an update message). The directory updates are scheduled locally in response to receipt of an update or copyupdate message. Our requirements state that this scheduling must be consistent with the partial ordering on actions. This is accomplished using the version numbers. Each split or merge changes the version numbers of the affected buckets. A split generates two buckets with version numbers one greater than that of the original bucket. A merge results in one bucket with a version number one larger than the maximum version of the two partners. The partial ordering is determined from the buckets and resulting version number associated with each action. For example, consider the following set of actions applied to the hash file in Figure 3 where the format for an individual action is <type of action and transaction number, first bucket involved, second bucket, resulting version number>:

{<split 1, bucket a, bucket d (new), version 2>
<split 2, bucket c, bucket e (new), version 3>
<split 3, bucket e, bucket f (new), version 4>
<merge 4, bucket d, bucket a, version 3>
<merge 5, bucket b, bucket a, version 4>}.

Then, using $<$ for the precedence relation,

split 1 $<$ merge 4 $<$ merge 5 and split 2 $<$ split 3.

Each directory manager schedules updates on its copy based on its record of which actions have already been incorporated into the structure. This information is encoded as version numbers in each entry of the table to be matched against the version number of updates (data supplied in the update or copyupdate message). Specifically, the Boolean function, VersionsDoNotMatch, must calculate the indices of the affected directory entries (using the pseudokey, whether the operation was an insertion or a deletion, and the local depth of the buckets prior to modification) and then compare version numbers of the entries and the message.

The requirement that deleted buckets remain available until all schedules contain the associated remove action is enforced by a conservative approach. The directory manager initially contacted for a request to delete that causes two buckets to merge is responsible for determining when the space can be reclaimed. It must collect acknowledgements related to the merge from all other directory managers and wait until transactions using old information from its own copy have finished before the partner's page can be deallocated. In fact, the directory manager waits for *all* outstanding acknowledgements and a quiescent local state before triggering garbage collection. Other directory managers wait until there are no transactions using their copies before sending acknowledgements for deletions.

The next step is to assume a well-behaved global directory and show that concurrent transactions do not interfere with each other or destroy the data structure.

First, we need to demonstrate that the search phase of a transaction arrives at the right bucket. The user's request for an operation may be directed to any available directory manager. In servicing this request, the manager generates a transaction number, decides which bucket manager to contact, and saves some state about the transaction. The information used to determine the appropriate bucket manager may be out of date because of insert or delete operations that are still in progress (i.e. the associated update or copy update message has not yet been processed).

Imagine a searching transaction that indexes into the directory and finds a pointer to bucket A as that directory entry is about to be changed to reflect a split or merge. If A has recently been split, A's next link will lead to the new bucket which contains the records moved from A. If A has just been merged into its partner, it will be marked as deleted, making it the "wrong bucket" for any search and the next link again will provide recovery. The important observation is that obsolete directory entries that are still visible always point to a bucket from which the correct bucket is reachable via next links. The changes in the bucket structure appear as atomic actions to concurrent transactions. In our formulation of the bucket manager, a slave process is spawned for each transaction within each manager involved in the transaction. Thus there is the need for locking to control concurrent access to a manager's buckets. Adding or removing a key without causing restructuring is done in a single disk put operation. If the target bucket for an insertion is full, it will be replaced by a pair of buckets in which the old contents are distributed between the two according to pseudokey. The new record will be included in the appropriate partner if there is room. The second half of the pair is written first in a newly allocated disk page and then the old bucket is replaced by the first half of the pair. Immediately after the first put, the new bucket is still not reachable through pointers in the hash file. Thus writing the pair is equivalent to the single operation of writing the first partner. Two buckets that are being merged are protected with exclusive-locks so intermediate states are not visible. Upon arriving at the right bucket, a process performing an insert or delete must also see the right version of it. Again a lock which excludes other updaters is required in order to read the bucket contents into private storage and is held until the bucket is rewritten (or it is discovered that no change is needed). Thus previous updaters have made their modifications known by the time a new updater gains its lock. Processes executing the find operation may legitimately see either an old or the new version of the target bucket.

Next, we consider potential inference among update transactions. Once an update arrives at the right bucket and gains the locks it requires, the actual modifications are essentially serialized. Thus updaters work with the most recent version of that bucket. However, for a deleter to get to the point where it has all the locks its needs can be somewhat involved if the target bucket is the "1" partner of a potential merge. The deleter must release its lock on the target bucket, place a lock on the "0" partner, and then re-lock the "1" partner. While this is taking place, other update operations may be affecting these buckets. In particular, a concurrent insertion could add new records to the target bucket once the deleter's lock is released so that it is not longer empty enough to allow merging. It is even theoretically possible for a stream of inserters to fill up the target bucket and cause a split, thereby moving the key that is to be deleted. In addition, another deleter might get the two partners locked and merged before the deleter we are focusing on does. Each of these conditions is checked for and the pitfalls avoided. After gaining the lock on the "0" partner, the deleter checks whether merging might be possible (the partner's next link points to the target bucket), and if this check fails, it goes back to simply trying to remove its key. If the two buckets are not linked in this way, it may mean the localdepths do not match or that the target bucket has been deleted. Attempting to lock the target bucket under these circumstances would carry with it the danger of deadlock. Upon finding the two buckets directly linked and re-locking the "1" partner, the deleter checks the emptiness of the bucket, whether the desired key is still there, and whether localdepths still match before going ahead with the merge. Unless the key has moved, the deleter at this point would have the needed locks and no further interference could occur at the bucket level.

Bucket manipulations that are completely contained within one bucket manager work almost exactly like the centralized solution [Ellis 83]. Processing may go outside the boundaries of one bucket manager if the search phase has arrived at the wrong bucket manager, a split is required and no space is available locally, or a merge appears necessary and the partner is remote. In each of these situations, a second bucket manager becomes involved. In this presentation of the algorithm, an off-site split is handled directly by the front end process since it does not affect existing buckets in the second manager's partition. For the other cases, another slave is created for the transaction by the second manager. A wrongbucket message transfers the necessary state for continuation of processing at the new site. Calls to SendAck and SendBucketdone generate messages that trigger the releasing of read-locks. If a split is called for, two or three processes (i.e. the originating directory manager, the bucket manager slave currently responsible for the full bucket, and possibly a bucket manager front end with available space) become involved; however, there is no real parallelism among them so the order in which the disk operations take place is well-defined.

The merge is slightly more complex. There are two cases to consider based on which of the partners the original bucket manager has. The Mergedown message and its associated reply are used when the first manager has the "0" partner of the potential merge to share state values needed by the other manager (e.g. the localdepths of the two buckets must be compared and new links must be set up in both buckets). The Mergeup protocol (i.e. Mergeup, MergeUpReply, and GoAhead messages) serves to exchange the information needed for the extra checking on mergability described above. Parallelism is allowed between the two bucket managers; however, because of the exclusive-locks protecting the two partners, the ordering of disk operations does not matter.

The freedom from deadlock argument depends on the fact that locks are requested according to an ordering on the buckets. While a bucket is locked, additional locks are requested only on buckets reachable from it via next links. Given the way deleted buckets are handled, it is not true that the ordering between two buckets stays the same for as long as both exist. Thus, initially bucket B may be reachable from bucket A but if they are partners this relationship may be reversed as B is merged into A. However, it is not possible for transactions following the old ordering to coexist with ones following the new ordering because during deletion exclusive-locks are used to ensure that all the slave processes with old information have cleared out of the vicinity of the merge. Extra precautions must be taken by the slaves involved in a deletion to check that the locking of partners is consistent with reachability.

This distributed implementation not only has locking as a potential source of deadlock but also involves message flows and internal scheduling of requests within managers. It is necessary to demonstrate that these factors do not introduce deadlock. A transaction could be blocked if it requires service from a process that is blocked on a receive message primitive or it is stuck in one of a directory manager's scheduling tables.

First, consider the message flows. Ignoring name lookup for ports, there is a single receive point in the directory manager code (in the procedure GetMessage at the top of the outer loop) and it accepts any incoming message regardless of message type or identity of sender. Basically the same statement holds for the bucket manager front end processes. Each instance of a bucket slave is dedicated to one transaction. This fact simplifies the analysis of protocols between bucket managers. For each receive point in the bucket slave code, we can characterize the state of both the sender and receiver. For example, the receivemessage in the find case is executed only when onmachine = false and SendWrongbucket has been done. This imples that messageid = Wrongbucket in the other slave process and SendAck is eventually executed. It is easy to see that the message flows through bucket managers do not cause deadlock by doing this kind of analysis for each receive point.

There are four ways in which a transaction can get delayed within directory managers: it may be in the context table awaiting a bucketdone or update message from a bucket manager, its directory updates may be delayed until versions match, copyupdate acknowledgements for deletions may be waiting for the equivalent of local exclusive−locking (i.e. a readcount of zero), and the initiation of garbage collection may be waiting for the analogue of global exclusive−locking (i.e. local exclusive−locking plus receipt of outstanding copyupdate acknowledgements). The first case presents no problem as long as a bucketdone or update message is sent back to the originating directory manager for each find, insert, or delete message. This is true as can be seen by following each branch of the bucket slave code for handling the find, insert, and delete message types. The second case requires a guarantee that versions eventually do match. The update message contains the old version numbers and the oldlocaldepth of the two buckets involved. The oldlocaldepth and the pseudokey are used to determine which directory entries must have the matching version numbers. The basis of the argument that the desired pattern of version numbers eventually occurs is the partial ordering on transactions previously described and the way this partial ordering is implemented using the version numbers. The ordering on transactions affecting shared buckets precludes two transactions each waiting for the other to advance a version number in the directory. The third and fourth sources of delay are related. The key observation is that readcount represents the number of transactions initially handled *locally* that have not yet applied their modifications to the *local* copy of the directory. Copyupdates are not reflected in the value of readcount. It is possible for readcount to reach zero if new requests do not continually arrive since delayed updates are not permanently blocked. Copycount becoming zero at some directory manager depends on each directory manager independently reaching the point where it is finished with all but the garbage collection work of the transactions it is responsible for. Thus, sending remembered acknowledgements and garbage collection can be indefinitely postponed by a steady stream of new requests but deadlock among a fixed set of transactions is not possible.

3. Incorporating Fault Tolerance

The solution just described does not address the issues of crash tolerance and recovery. The structure of that solution reveals that it is a fairly straightforward adaptation of the earlier concurrent algorithm. Consideration of crash recovery suggests a slightly different organization.

The problems associated with processor and communication failures could be conveniently avoided if it were possible to embed updates to the hash file in a system based on atomic transactions. However, the goals guiding the design of our solution (e.g. concurrency and availability) have led to locking protocols that are not compatible with standard commit protocols. As we shall see, the atomic transaction construct is a useful tool when applied to small groups of steps within the processing of an individual update operation.

The kinds of failures being addressed include the failure of a manager with loss of all associated volatile process state but not of its portion of the hash file residing on disk. Lost messages and network partitions are also considered. We assume that it is possible to detect the death of a process. The IPC mechanism used here as the model of communication provides notification to potential senders when a port disappears (for example as a consequence of the receiver's death) so this assumption is reasonable.

The most significant problems with the current distributed solution have to do with interactions among the directory servers. In particular, directory updates are funneled through the one directory server initially contacted for the operation and it forwards copyupdate messages to all the other directory managers. A directory manager can not allow the garbage collection of the set of deleted buckets for which it is responsible until it has collected acknowledgements from all other directory managers. Furthermore, if a server goes down before propagating the directory update information, the scheduling of other updates at other managers is affected.

In order to prevent a failed directory manager from holding up completion of an operation, we need the ability to remove unavailable servers from participation in the normal directory update routine and re-enlist them later. Thus, an individual bucket may be deallocated when all directory servers either acknowledge the associated update message or are designated as being down. This approach requires additional information in acknowledgement messages (i.e. identification of transaction and sender) and more state kept by the server responsible for outstanding (not yet fully acknowledged) directory updates. A directory manager that is rejoining the system must construct a sufficiently up-to-date copy of the directory before resuming normal processing. The key observation is that the buckets contain the necessary information for building such a copy (i.e. localdepth, commonbits, and the next link). The starting point for a scan of buckets, namely the first bucket, is in a fixed location and never moves during restructuring; so any old version of the directory can be used to find it. The manager follows next links through all the buckets using a lock-coupling protocol with selective-locks. This approach leaves the recovering manager vulnerable to failures of bucket managers. If this possibility is determined to be unacceptable, the manager can start with a reasonably good copy of the directory (acquired from a healthy directory server) and use the bucket scan to verify and update its entries. In this case, upon encountering an unavailable bucket manager, the server can take a fairly low-risk chance of missing some information and skip over those buckets.

The second major aspect of this more fault tolerant solution is a reassignment of responsibilities. Rather than having the propagation of directory updates handled by one of the directory managers, the bucket manager in charge of the bucket update broadcasts the directory update messages and collects the acknowledgements. Since the directory manager initially contacted does not assume responsibility for an operation once the appropriate bucket manager has the target bucket locked, the bucket manager can immediately send a bucketdone message to allow the directory manager to forget any state it had saved about the transaction and later send it an update message if necessary. The bucket managers must maintain a list of directory managers believed to be up. The bucket scan performed by a recovering directory manager serves to announce its existence to the bucket managers. In addition, bucket managers can periodically exchange their up-lists. Recovering bucket managers must acquire a current up-list and send information to directory managers for verification of their entries for its buckets. The removal of a network partitioning is detected during the exchange of up-lists and dealt with by the bucket manager recovery mechanism.

The advantage of this reorganization is that the failure of a bucket manager prevents subsequent updates concerning those buckets from occurring so the fact that the directory updates may not get sent is not as much of a problem as it is when one of the directory managers is supposed to send those update messages and it is down. In that case, there is more potential for subsequent bucket operations whose associated directory updates will be held up by the missing messages.

The remaining details needed for fault tolerance are applications of standard techniques such as timeout and retransmission of messages. The act of writing the two buckets involved in a merge operation back to disk should be done atomically using a commit protocol. The order of writing the two buckets involved in a split operation makes them visible in one atomic step but failures during this action may result in the allocation of a new bucket that never gets incorporated into the data structure. It may be convenient to enclose the disk operations involved with splitting within an atomic action as well.

Figure 9 shows the revised message flow for the increased degree of fault tolerance provided. Figures 10 and 11 give the pseudocode for the managers implementing this solution.

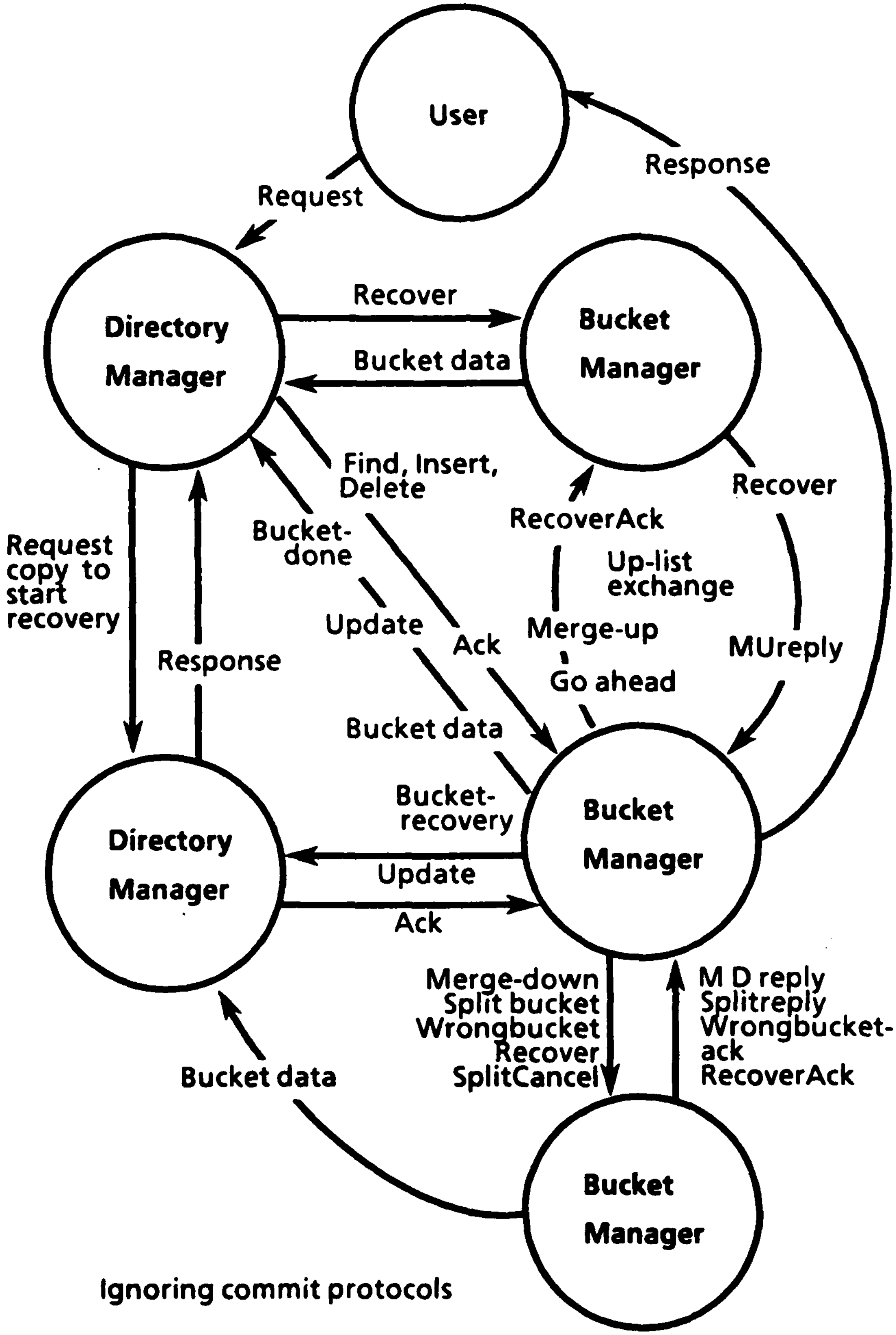


Figure 9 Fault Tolerant Messages

Figure 10 Pseudocode for Fault Tolerant Directory Managers

```
while ( true ) {
    messageid = GetMessage (&msg);
    /* Either receives a message or takes a message off
    the list of delayed but now ready directory updates.
    Messageid = "timeout" if list is empty and receive primitive
    times out. */
    switch (messageid) {
        case request: /* from user */
            Calculate pseudokey and locate current
            incarnation of the bucket manager responsible
            for desired bucket;
            /*Maps bucket manager id to port either through
             a local cache or the IPC name server*/
            if (valid port found) {
                readcount = readcount + 1;
                /*number of transactions
                currently using directory data*/
                Generate transaction # and save state related
                to this request;
                Set timer for transaction #;
                Construct and send a "find", "insert" or
                "delete" message to bucket manager;
            }
            else send user a failure reply;
            break;

        case bucketdone: /* from bucket manager */
            if (transaction # not in use)
            /*This is a duplicate message or state lost in crash*/
                break;
            readcount = readcount - 1;
            CleanState (msg.transaction #);
            /* forget about this request */
            break;

        case update: /* from bucket manager */
            if (VersionsDoNotMatch(msg))
            /* compares version numbers in message with
            version numbers in corresponding directory entries:
            detects duplicate update messages that have already
            been done and reissues acks in appropriate way */
                DelayUpdate(msg);
                /*Eliminates duplicates of messages in queue*/
            else {
                if (msg.op = = insert) {
                    Apply appropriate updates
                    to local copy of directory;
                    SendAck(msg.ackport, transaction #, myport);
                     /*respond to bucket manager
                       who initiated this update */
                }
                else { /* op = delete */
                    Apply local directory updates;
                    RememberAck(msg.ackport, transaction #, myport);
                    /* save up acks until the equivalent of
                    exclusive-locking occurs */
                }
                ReleaseSaved();
                /*If finishing this directory update enables previously
                delayed ones, make them accessible to GetMessage */
            }
            break;

        case reinit: /*from OS's process manager that
        restarted directory manager process.
        Note this uses a conservative approach-
        doesn't skip bucket managers*/
            do {
                Contact IPC name server to locate a healthy
                directory manager and
                send a "request copy" message to it;
                Receive response containing copy of directory;
            } while (timeout or emergency message received).
            NoGoodMessage = true;
            while (NoGoodMessage) {
                lookup port to first bucket manager;
                if (no valid port found) delay().
                else {
                    send "recover" message;
                    while (true) {
                        Receive message;
                        if (timeout) break;
                        if (it is a bucketdata message) {
                            NoGoodMessage = false;
                            break;
                        }
                        if (emergency message about
                          this port) {
                           delay();
                           break;
                        }
                        /*ignore irrelevant emergency
                        message or duplicate directory copy*/
                    }
                }
            }
            /*falls through to next case*/

        case bucketdata: /*from bucket managers
        in response to recover message*/
            Update directory entry with information in message;
            if (all bucket managers have replied)
            /*all directory entries have been verified*/
                Publicize own named port with name server;
                /*Implies this manager now has healthy copy-
                 and will now serve users' "request" messages
                 and recovering directory managers' "request
                 copy" messages (accesses gotten through
                 name server)*/
            break;

        case bucketrecovery: /*from recovering bucket manager/*
            Update directory entries, if necessary, to reflect
            true state of buckets;
            Cache port of bucket manager in local name table;
            break;

        case emergency: /*from IPC - notification of port death*/
            Remove port from cached name table;
            Place an indicator in state of each associated
            transaction that initially contacted port has died;
            /*doesn't just abort transaction since another
            bucket manager may now be involved (wrongbucket protocol)
            so waits until timer for transaction goes off */
            break;

        case timeout: break;

    }
    if (!readcount) SendRememberedAcks();
    /*send acks saved by deletion updates */
    for (all transactions, t, whose timers have expired){
        RestoreState(t);
        if (t.portdied)
            Send user a failure reply;
        else Retransmit;
    }
}
```

Figure 11 Pseudocode for Fault Tolerant Bucket Managers

Bucket Manager Front End Process:

```
/*Note that communication between bucket managers involves
IPC name lookup- the presentation here generally assumes that
a valid port is found*/

while (true) {
    messageid = receivemessage (&msg);
    switch (messageid) {
        case splitbucket: /* from another bucket
        manager with no available space */
            Allocate available page;
            putbucket (newpage, msg.half2);
            /*as written, failure here makes newpage garbage*/
            Send "SplitReply" message containing link to new bucket;
            break;
        case cancelsplit: Deallocate page assigned;
            break;
        case recover:
            Update up-list;
            Broadcast revival to existing slaves;
            Create slave to handle response;
            Forward message to it;
            break;
        case emergency message about slave:
            Update transaction # - slave table;
            break;
        case reinit: /*from O.S.'s process manager*/
            /*Get an up-list from a neighboring bucket manager-
             a "neighbor" being another bucket manager connected
             via next or prev links from locally managed buckets-
             front end process probably should maintain
             a cache of bucket managers id's - current port if known*/
            while (true) {
                Locate port, p, for one of neighboring
                bucket managers;
                Send "empty" up-list to initiate up-list exchange;
                messageid = receivemessage (&msg);
                    /*with finite timeout*/
                if (messageid = = up-list exchange) break;
            }
            for (all directory managers in msg.up-list)
                Send "bucketrecovery" message;
            MergeUp-lists(&up-list, msg.up-list);
            Allocate public port and assert my long-term name for it;
            break;
        case up-list exchange:
            if (not a reply) Send own up-list in response;
            MergeUp-lists (&up-list, msg.up-list);
            for (all directory managers in msg.up-list - up-list) {
                Send "bucketrecovery" message;
                Broadcast revival to existing slaves;
            }
            break;
        case timeout:
            Initiate up-list exchange with all healthy neighbors;
            /*don't worry if valid port can not be found for one*/
            break;
        default:
            if (transaction # not yet seen) {
                Create a bucket slave process and forward msg to it;
                Record transaction # - slave mapping;
            }
            else { /*duplicate message*/
                lookup transaction #;
                if (associated slave still alive)
                    forward message;
                else Send appropriate reply;
            }
    }
}
```

Bucket Slave Process:

```
messageid = receivemessage (&msg);
/*Includes data needed to initialize local copy of up-list*/
if (messageid = = wrongbucket) sw = msg.op;
else sw = messageid;
switch (sw) {

    case recover: /*Could try to package information about consecutive
        buckets in one message, but doesn't in this version*/
        oldpage = msg.page;
        SelectiveLock (oldpage);
        Send "recoverack" to msg.replyport;
        getbucket (oldpage, current);
        Construct and send "bucketdata" to recovering directory;
        onmachine = true;
        while (onmachine) {
            newpage = current -> next;
            machine = current -> nextmgr;
            if (machine = = nil) break;
            if (machine != me) {
                Send "recover" to next manager;
                GetRecoverAck(); /* loops until achieved,
                retransmits if timeout, WAITS if destination
                known to be down*/
                onmachine = false;
            }
            else {
                SelectiveLock (newpage);
                getbucket (newpage, current);
                UnSelectiveLock (oldpage);
                Send "bucketdata";
                oldpage = newpage;
            }
        }
        UnSelectiveLock (oldpage);
        ClearDuplicates();
        break;

    case find: oldpage = msg.page;
        ReadLock (oldpage);
        if (messageid = = wrongbucket)
            Send "Ack" to bucket manager holding
            previous bucket; /*allows it to unlock */
        else Send "Bucketdone" message to directory manager;
        getbucket (oldpage, current);
        onmachine = true;
        /*Follow next links until current is the right bucket:*/
        while (current is wrong bucket && onmachine) {
            newpage = current -> next;
            machine = current -> nextmgr;
            if (machine != me) { /* next bucket is remote */
                Send "Wrongbucket" message to next bucket
                manager;
                onmachine = false;
            }
            else { /* next bucket is local */
                ReadLock (newpage);
                getbucket (newpage, current);
                UnReadLock (oldpage);
                oldpage = newpage;
            }
        }
        if (onmachine) {
            if (search (current, msg.key))/* is key there? */
                found (msg.key);
                /*Send user response indicating key found*/
            else
                notfound (msg.key);
        }
        else GetWrongbucketReply(); /*See below*/
        UnReadLock (oldpage);
        ClearDuplicates(); /*See below*/
        break;
```

```
case insert: onmachine = true;
    oldpage = msg.page;
    SelectiveLock (oldpage);
    if (messageid = = wrongbucket)
        Send "Ack" to previous bucket manager;
    else Send "bucketdone" message;
    getbucket (oldpage, current);
    Follow next links until current is the right bucket
    (as in find case except use Selective locks instead of
    Read locks);
    if (!onmachine) {
        GetWrongbucketReply();
        UnSelectiveLock (oldpage);
    }
    else {
        if (search (current, msg.key)) {/*is key already there?*/
            success = true;
            UnSelectiveLock (oldpage);
        }
        else if (current -> count != numentries) {
            /* current bucket not full */
            success = true;
            add (current, msg.key);
            /*inserts key into current buffer */
            putbucket (oldpage, current);
            UnSelectiveLock (oldpage);
        }
        else {/*current is full - directory will be affected */
            success = split (current, half1, half2, msg.key);
            /*distributes the contents of the current bucket into
            2 buffers pointed to by half1 and half2;
            if room available, inserts key into appropriate half
            and returns true; otherwise returns false */
            if (AvailablePages()) {
                newpage = allocbucket ();
                machine = myid;
                putbucket (newpage, half2);
            }
            else {/* no available pages locally */
                done = false;
                while (not done) {
                    Send "Splitbucket" message
                    containing contents of new bucket
                    to any manager with space;
                    while (true) {
                        messageid = receivemessage (&msg);
                        if (timeout) break;
                        if (messageid = = splitbucketreply) {
                            done = true;
                            break;
                        }
                        if (emergency message about
                        potential partner) break;
                        if (messageid = = wrongbucket)
                            Send "ack";
                        if (messageid = = request)
                            Send "bucketdone";
                        if (emergency message about
                        death of some directory
                        manager or message from
                        front end about recovery
                        of one) update up-list;
                    }
                }
                machine = msg.bucketmgr;
                newpage = msg.page;
            }
            half1 -> next = newpage;
            half1 -> nextmgr = machine;
            /*as written, failure prior to putbucket
              makes newpage garbage that won't get cancelled*/
            putbucket (oldpage, half1);
            UnSelectiveLock (oldpage);
            BroadcastUpdates(); /*See below*/
        }
        if (success)
            Send user response;
        else
            Send "request" message to any directory
            manager as if it came from user;
    }
    ClearDuplicates();
    break;

case delete: needupdate = false;
    success = true;
    Find the right bucket as in the beginning of insert
    except place Exclusive locks;
    if (!onmachine) {
        GetWrongbucketReply();
        UnExclusiveLock (oldpage);
    }
    else {
        if (current bucket will not be left "too empty"
          as a result of deleting msg.key) {
            if (remove (msg.key, current))
                putbucket (oldpage, current);
            UnExclusiveLock (oldpage);
        }
        else { /*Merging partner buckets is called for*/
            if (msg.key is in first bucket of the pair) {
                newpage = current -> next;
                machine = current -> nextmgr;
                if (machine = me) {
                    Merge on site;
                }
                else {/* partner is remote */
                    Send "Mergedown" message to
                    partner's bucket manager.
                    /* Mergedown Reply expected */
                    while (true) {
                        messageid = receivemessage (&msg);
                        if (timeout) {
                            msg.success = false;
                            break;
                        }
                        if (messageid = = MergedownReply) break;
                        if (emergency message about partner's
                          bucket manager){
                            msg.success = false;
                            break;
                        }
                        Deal with possible duplicates;
                        /*as in GetWrongbucketReply*/
                    }
                    if (msg.success) {/*OK to merge
                    (i.e. localdepths match);
                    contents of partner in msg */
                        Construct merged bucket
                        in current buffer;
                        Start atomic action with partner;
                        putbucket (oldpage, current);
                        End atomic action; /*Commit protocol*/
                        if (aborted) success = false;
                        /*if committed, partner will be responsible for
                          propagating "update" messages*/
                    }
                    else {/* simply remove record */
                        if (remove (z, current))
                            putbucket (oldpage, current);
                    }
                    UnExclusiveLock (oldpage);
                }
            }
```

```
else { /* msg.key in second of pair */
    newpage = current -> prev;
    machine = current -> prevmgr;
    UnExclusiveLock (oldpage);
    if (machine = = me) {
        Merge on site;
    }
    else {/* partner is remote */
        Send "Mergeup" message to
        partner's bucket manager;
        /* MergeUp Reply expected */
        while (true) {
            messageid = receivemessage (&msg);
            if (timeout) Retransmit "Mergeup" message;
            if (messageid = = MergeUpReply) break;
            if (emergency message about
                partner's bucket manager) {
                msg.success = false;
                break;
            }
            if (emergency message about
                death of some directory manager or
                message from front end about
                recovery of one) update up-list;
            Deal with duplicates;
        }
        if (!msg.success) {
        /* not mergable - simply remove record */
            if (remove (z, current))
                putbucket (oldpage, current);
        }
        else {/* apparently mergable from
        partner's point of view- check more locally */
            ExclusiveLock (oldpage);
            getbucket (oldpage, current);
            if (key to be deleted no longer
            belongs in current bucket) {
                UnExclusiveLock (oldpage);
                Send "Goahead" message to partner
                with success field set to false;
                /*cancels merge */
                success = false;
            }
            else if (current->localdepth
                does not match localdepth in msg ||
                current no longer "too empty") {
                    if (remove(z, current))
                        putbucket (oldpage, current);
                    UnExclusiveLock (oldpage);
                    Send "Goahead" message
                    with success = false;
                    /*cancel merge */
                }
                else {
                    Send successful "Goahead"
                    message to partner;
                    /*tell partner's manager to merge */
                    current -> next = current -> prev;
                    current -> nextmgr =
                        current -> prevmgr;
                    current -> commonbits = deleted;
                    Start atomic action with partner;
                    putbucket (oldpage, current);
                    End atomic action;
                    if (committed) needupdate = true;
                    else success = false;
} } } } } } }
```

```
            if (success)
                Send user resonse;
            else
                Send "request" message to any directory
                manager as if it came from user;
            if (needupdate) {
                BroadcastUpdates();
                ExclusiveLock (oldpage);
                deallocate (oldpage);
                UnExclusiveLock (oldpage);
            }
        }
        ClearDuplicates();
        break;

    case mergedown: newpage = msg.partner;
        ExclusiveLock (newpage);
        getbucket (newpage, brother);
        success = brother -> localdepth = = msg.localdepth;
        Send "MergeDown Reply" to partner;
        if (success) {
            brother -> commonbits = deleted;
            brother -> next = brother -> prev;
            brother -> nextmgr = brother -> prevmgr;
            Start atomic action with partner;
            putbucket (newpage, brother);
            End atomic action;
            if (committed) {
                UnExclusiveLock (newpage);
                BroadcastUpdates();
                ExclusiveLock (newpage);
                deallocate (newpage);
            }
        }
        UnExclusiveLock (newpage);
        break;

    case mergeup: newpage = msg.partner;
        ExclusiveLock (newpage);
        getbucket (newpage, brother);
        success = (brother -> next = = msg.target) &&
        (brother -> nextmgr = = msg.managerid);
        Send "MergeUp Reply";
        if (success) {
            /* "GoAhead" expected */
            while (true) {
                messageid = receivemessage (&msg);
                if (timeout) msg.success = false;
                if (messageid = = GoAhead) break;
                if (emergency message about partner's manager) {
                    msg.success = false;
                    break;
                }
                if (messageid = = MergeUp) /*duplicate*/
                    Retransmit "MergeUpReply";
            }
            if (msg.success) { /* merge */
                Construct merged bucket in brother;
                Start atomic action with partner;
                putbucket ( newpage, brother);
                End atomic action;
            }
        }
        UnExclusiveLock (newpage);
        ClearDuplicates();
        break;
}
```

```
BroadcastUpdates()
{
    notdone = true;
    while (notdone) {
        Send "update" messages to all
        directory managers on up-list;
        while (true) {
            messageid = receivemessage(&msg);
            if (timeout) break;
            if (messageid = = updateack)
                Mark that manager on up-list;
            if (relevant emergency message)
                update up-list;
            if (recovered directory manager) {
                update up-list;
                Send "update" message;
            }
            if (messageid = = wrongbucket) /*duplicate*/
                Send "ack";
            if (messageid = = request) /*duplicate*/
                /f3Send "bucketdone";
            if (all directory managers accounted for) {
                done = true;
                break;
            }
            /*Ignore irrelevant emerg. msgs*/
        }
    }
}
```

```
GetWrongbucketReply()
{
    while(true) {
        messageid = receivemessage (&msg);
        if (timeout) Retransmit "wrongbucket" message;
        if (messageid = = WrongbucketReply) break;
        if (emergency message about next bucket manager) {
            send user a failure response;
            break;
        }
        if (messageid = = wrongbucket) /*duplicate*/
            Send "ack";
        if (messageid = = request) /*duplicate*/
            /f3Send "bucketdone";
    }
}
```

```
ClearDuplicates()
{
    while (any messages pending) {
        messageid = receivemessage (&msg);
        switch (messageid) {
            case wrongbucket: Send "ack";
            case request: Send "bucketdone";
            case splitbucketreply: /* only possible in insert*/
                if (from other than chosen partner)
                    Send "cancelsplit";
            /*Other possible duplicates
              (e.g. Mergeup, MergeUpReply)
              require no action*/
        }
    }
}
```

4. Conclusions

In this paper, we have presented a solution for distributing an extendible hash file with replication of the directory component of the structure. The solution is interesting in its own right for use in a distributed data base system that is expected to frequently change size and be available from various points in the network.

The solution also serves to illustrate several points that may apply to other problems that can be viewed primarily as data structures to be partitioned and possibly replicated across sites of a distributed system. The first point concerns the features of sequential data structures that make them amenable to distribution. In this study, we chose a shallow (2-level) linked structure as a starting point. For comparison, we can consider the deeper structure such as a B-tree or a logically contiguous one such as linear hashing [Litwin 80]. First of all, a multilevel linked structure offers several advantages. The links map naturally onto a port-based communication mechanism and the indirection provided by the directory allows flexibility in assigning buckets to sites. It is especially convenient if the top level component is reasonably small. In our case this allows the hash function to calculate a location in the addresss space belonging to a single logical processor. By contrast, linear hashing lacks the directory component and therefore requires that a naming convention be adopted to give the appearance of a network-wide address space appropriate for direct calculation of bucket locations. Of course, if the directory outgrows a single manager, extendible hashing requires a similar convention.

The major complexity of our solution arises from the replication of the directory to enhance availability. Although the absence of a directory in the linear hashing scheme seems at first glance to provide availability easily, there is a small set of data required for bucket address calculation that should be replicated. In the naive solution, this information should also be accurate, suggesting a need for strict synchronization among copies. Thus eliminating the directory component does not trivialize the problem as some researchers have claimed. The shallowness of our multilevel structure is an asset in that the short average search path makes an optimal assignment of buckets to managers relatively unimportant. For a deeper structure such as a B-tree, one might want to address the hard problem of grouping pages within servers to improve locality.

The second point demonstrated by our solution is the value of making modifications in the implementation of the data structure that allow recovery from the use of inconsistent information (e.g. next links) and improved locality (e.g. prev links). There are opportunities for taking this idea even further in the solution presented.

Another point has to do with methodology. Developing a distributed solution raises a number of issues; although some are unique to this particular model of computation, the aspect of achieving a degree of concurrency is common to both distributed and shared data systems. Thus a correct centralized solution should prove to be a good starting point in determining how to partition structured data. The approach successfully used here was to first solve the problem of concurrent access and then use that result as the basis for distributing the computation.

Finally, it bears repeating that a fundamental characteristic of a distributed system is the impracticality of gathering a true instantaneous global view of the world. Successful distributed applications must be able to accommodate inconsistent and inaccurate information.

5. References

[Ellis 83] C. Ellis,
"Extendible hashing for concurrent operations and distributed data," *Proceedings*, 2nd SIGACT-SIGMOD Symp. on Principles of Database Systems, March 1983.

[Fagin 79] R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong,
"Extendible hashing - A fast access method for dynamic files," *ACM TODS*, Vol. 4, No. 3, September, 1979, 315-355.

[Gifford 79] D. Gifford,
"Weighted voting for replicated data," *Proceedings*, 7th Symp. on OS Principles, December, 1979.

[Kernighan 78] B. Kernighan and D. Ritchie,
The C Programming Language, Prentice-Hall, 1978.

[Larson 78] P.A. Larson,
"Dynamic hashing," *BIT*, Vol. 18, No. 2, 1978, 184-201.

[Lehman 81] P. Lehman and S.B. Yao,
"Efficient locking for concurrent operations on B-trees," *ACM TODS*, Vol. 6, No. 5, December 1981, 650-670.

[Litwin 78] W. Litwin,
"Virtual hashing: A dynamically changing hashing," *Proceedings*, 4th Conf. Very Large Data Bases, 1978, 517-523.

[Litwin 80] W. Litwin,
"Linear hashing: a new tool for file and table addressing," *Proceedings*, 6th Conf. Very Large Data Bases, 1980, 212-223.

[Lomet 83] D. Lomet,
"Bounded index exponential hashing," *ACM TODS*, Vol. 8, No. 1, March 1983, 136-165.

[Rashid 80] R.Rashid,
"An interprocess communication facility for UNIX," CMU-CS-80-124, Carnegie-Mellon University, June 1980.

[Stonebraker 79] M. Stonebraker,
"Concurrency control and consistency of multiple copies of data in distributed INGRES,"
IEEE Transactions on Software Engineering, Vol SE-5, No. 3, May 1979.

[Thomas 79] R.H. Thomas,
"A majority consensus approach to concurrency control for multiple copy databases,"
ACM TODS, Vol. 4, No. 2, July 1979, 180-209.

END
FILMED

5-86

DTIC